# Lab 1: R Tutorial

Nora Mitchell

January 2017

## Contents



# 1 Introduction

R is a powerful, open-source program used for statisics and graphics. R uses its own programming language and base, but much of its power lies in special "packages" that users can write to perform detailed, field-specific analyses. R can also communicate with other programs (such as JAGS), making analyses easier to run on both PC and Mac systems.

RStudio is one of a number of programs which interface with R (and the one I will be using throughout the course). Feel free to use your own if you wish.

### 1.1 Installing R

R can be installed from `http://www.r-project.org/`. You will have to select a specifc CRAN (Comprehensive R Archive Network), scroll down to USA and select the Statlib one from Carnegie Mellon. Choose the appropriate download for your OS, and install either the "base" R v3.1.2 for Windows or the correct version for your Mac OS.

### 1.2 Installing RStudio

You can now open R, but you will see only a console window. You can type and run lines of code in here, but it is not user friendly and is difficult to work and save materials in. Therefore, we wil install RStudio.

Download RStudio (**open-source desktop version**) at `http://www.rstudio.com/products/rstudio/download/`. Once you have done this, open RStudio. It should automatically communicate with R

You may wish to play around with the windows within RStudio. These include:

- main window "script": for editing text/writing code
- console: displays code actually run (plus output)
- a window displaying all variables, functions, values, etc. currently in use
- a window for viewing files in your working directory, plots, packages installed, and help documentation

## 2 Getting Started

Learning to use R takes a lot of time and love, but it is also incredibly valuable and the new standard in ecology and evolutionary biology. This tutorial will walk you through some of the very basics. A lot of learning R is looking at documentation for individual packages or doing Google searches.

If you're familiar with R, you may wish to skip this section.

### 2.1 Writing & Running Code

A few basics before getting started. We will be working in the main window writing our code.

- To actually run a line of code, you highlight the line and hit CTRL+Enter (or apple enter on a Mac), or click the "Run" icon. The line which has run, and any output, will then be displayed in the active console.

- To comment out lines (type in notes to yourself that won't actually be run), use the pound sign #, followed by any text. R will not read this as code. Hint: It's best to keep your code well-documented, so that you can come back to it months later and actually know what you did/what you were thinking.

- **functions** take a set of **arguments** (input by the user) and return an **object**. Functions can be pre-written and contain code that transforms the arguments into the output. You can also write your own functions (woo!)

- Help in R: If you have a question on a specific package or function, simply run a line of code with a ? and then the function's name. The documentation for that will open under the "Help" tab in RStudio.

  input: ?mean

  Check the help console: it will now have information about the function "mean".

## 2.2 Basic Math

Let's start very basic. Keep the same script (main window), as we may reuse some pieces in later examples. In your script, type

```
4+8
```

and hit CTR+Enter (or Run). In the console, it should now say

```
> 4+8
[12]
```

You can also store results as variables in R, which can be used later. You can use either = or <- for assignment. I typically prefer <-, since this has a broader scope (ask me for details if you want).

**Aside: Naming variables**

Variable names in R are case sensitive! Current convention is:

1. Start variable names with lowercase letters
2. Separate words in the variable names with underscores (separate_with_underscores) (periods.will.work.but.not.conventional)
3. Use only lowercase letters, underscores, and numbers in variable names

4. Can't start a variable name with a number

5. Can't use special symbols

6. Don't use the names of common functions

Now assign a value to a variable called "a"

```
a <- 15 + 16
a
```

Run both lines of code. You're output should look like:

```
[1] 31
```

You can also store multiple variables and then do math on them

```
b < -23
c <- 42
d <- b*c
d
```

Output:

```
[1] 966
```

**Challenge Question: height**

What is the output of the following?

```
height_inches <- 68
height_cm <- height_inches * 2.54
height_inches <- 75
height_cm
```

## 2.3 More Basic Functions

There are a number of built-in mathematical function in R, including constants, basic transformations, trigonometry, and different statistical distributions (which we will go over later in more detail).

| Basic Math in R | |
|---|---|
| abs($x$) | absolute value of $x$ |
| cos($x$), sin($x$), tan($x$) | cosine, sine, and tangent of angle $x$ |
| exp($x$) | exponential function |
| log($x$) | natural (base-e) logarithm |
| log10($x$) | common (base-10) logarithm |
| mean($x_1, x_2, ..., x_n$) | takes the average of included numbers |
| range(...) | returns the minimum and maximum of the given arguments |
| pi | constant $\pi$ |
| rnorm($x$) | $x$ random numbers from normal distribution N(0,1) |
| runif($x$) | $x$ random numbers from uniform distribution U(0,1) |
| sqrt($x$) | square root of $x$ |

## 2.4 Data types

R has different types of data that interprets and treats in certain ways. These are:

- double: typeof(3.14)
- integer: typeof(1L)
- complex: typeof(1+1i) (rarely used in ecology)
- logical: typeof(TRUE)
- character: typeof("banana")

## 2.5 Data Structures

R has several different types of data structures, with different attributes to them. We'll *very briefly* go over a few here. There is a TON that you can do with these, feel free to explore on your own.

- **Vectors** are 1-dimensions arrays of numbers, and can be used in calculations like a normal number

  input: x <- c(1:8)
  x
  output: [1] 1 2 3 4 5 6 7 8

  Here, the 'c' is a generic function that *combines* elements. '1:8' generates a list from 1 to 8, going up by 1. There are many other ways to make vectors, including '*seq*' and *'rep*' or typing them in by hand.

- **Matrices** are 2-dimensional arrays of numbers

  input: y <- matrix(c(1,2,3,4,5,6),2,3)
  y
  output: > y

```
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

  Here we've made made a matrix consisting of 6 values, with 2 rows and 3 columns. It does this by column automatically, but you can use byrow=T to change this

  input: z <- matrix(c(1,2,3,4,5,6),2,3, byrow = T)
  z
  output: > z

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

- **Factors** look like character data but represent categorical information. Data issues are often due to things being factors when they shouldn't be or vice versa. You can change data into factors using *as.factor* or into numerics using *as.numeric* if need be

```
x <- c(1, 2, 3, "a")
typeof(x)
x <- as.numeric(x)
typeof(x)
```

- **Data frames** are extremely common and useful. Data frames have multiple columns of equal-length vectors, and rows of individuals or observations. We'll explore some example datasets in data frame format in the next section. For now, here are some important functions we use with data frames:

| Data Frames | |
|---|---|
| data.frame($x, y, ...$) | combines $x, y$ into a data frame |
| names($x$) | access the column names in the data frame $x$ |
| as.data.frame($x$) | converts an object into a data frame |
| attach($x$) | includes columns of $x$ in the workspace |
| head($x$) | view first few rows of $x$ |
| tail($x$) | view last few rows of $x$ |
| summary($x$) | gives 5-number summaries and NA's for data frame $x$ |
| str($x$) | gives the internal structure of data frame $x$ |
| colnames($x$) | gives the column names of data frame $x$ |
| dim($x$) | gives the dimensions (rows and columns) of data frame $x$ |

Let's do a quick example of creating a data frame from 3 separate pieces on New Mexico Cities:

input:

```
city <- c("Albuquerque", "SantaFe", "LasCruces")
population_thousands <- c(600, 84, 102)
elevation_feet <- c(5300, 7200, 3900)
NM_cities <- data.frame(city, population_thousands, elevation_feet)
NM_cities
str(NM_cities)
```

Run all lines of code. You should be able to see that you've made a data frame with 3 columns and 3 rows: one column is a factors and two are numeric. To get the data from just one column, use the $and the column name. You can also use this to add a column name:

Hint: use tab completion to speed things up!

input:

```
NM_cities$population_thousands
NM_cities$mayor <- c("Keller", "Gonzales", "Miyagishima")
str(NM_cities)
```

You should see that the first line returns just the population, and then you've added a column for the mayor of each city.

str() is especially useful, because it tells you what kind of data type each column is (integer, numeric, character, factor, etc.)

**Challenge: Understanding Data Frames**

What is the structure of a data frame in R (what type of objects are the rows and

columns?)

## 2.6 Installing Packages

R comes pre-installed with the {base} package, but again, the sky's the limit! You have to install other packages. RStudio makes it especially easy to install packages in a few ways:

1. Use 'Tools' from the main toolbar.
   Go to 'Tools'-> 'Install Packages'. Search for a package by beginning to type its name. Here we'll install 'dplyr'. Make sure 'Install Dependencies' is checked–many packages depend on functions from other packages to work. Click 'Install' and it should work-though it may take a few seconds to finish.

2. Use the 'Packages' tab in the open window.
   In the 'Files, Plots, Packages, Help' window, select 'Packages'. Then click 'Install Packages', and follow the directions as above.

3. You can also use code, such as
   install.packages("dplyr")

Now you have the packages installed. To actually use them, you have to **call** them
    library(dplyr)
...and the package (and all of its functions) is ready to use!

# 3 Analyzing Data

## 3.1 Changing Working Directory

Download the .txt files from the webpage and place them in a folder of your choice on your laptop. In order to access these files, you have to navigate so that R has access to that **working directory** (folder). You will then have access to files in those folders, and any files you save will also be in that folder. To do this:

Go to 'Session'-> 'Set Working Directory' -> 'Choose Directory' and navigate to the folder with your files in it.

Your console will now show a shortened path to your folder
    setwd(" /SEV")

You can also set your working directory to 'Source file location', if you have saved your

R script in a particular location. You can also type in the *setwd* code itself and save it for when you open this R script again in the future. If you forget what working directory you're in, you can check using *getwd()*.

## 3.2 Importing Data

Now that you've navigated to the proper working directory, you can import data saved in various file formats, including text or tab-delimited files in .csv or .txt format, or more specific formats like .nex or .tre. We will be working mostly with .csv or .txt files, so I will use one of these as an example

To read in a text file, we can use

*read.table*("filename.txt", header=TRUE, sep = ",", na.strings="")

The header portions means that the first row of the data is a header, sep means that the data are comma separated, and finally missing values are indicated by whatever you input into na.strings = "". You can edit this if need be, but this is fairly standard. There are corresponding functions to read in other data file types, such as *read.csv*.

Let's read in an example data set from the Sevilleta on creosote. Let's use different dataframe functions to explore the data a little bit.

input: data <- *read.table*("sev024_creosotedimension_01112010_0.txt", header=TRUE, sep = ",", na.strings="-888")

Take a look at the "Workspace" console–you should now see several different Values and Data objects with brief descriptions. You can click on these and a new window will open to view the data.

Once your data are in a workable format, the possibilities are endliess. Let's investigate using a quick challenge

**Challenge: Creosote Data Info**

- How many observations are there?
- What data type is the "Species" column?
- How many observations are missing height and diameter?
- What are the earliest and latest years of data collection?

Real data are almost always complicated and messy. There are a ton of ways to analyze your data in R, especially using packages. Be prepared to learn on the fly!

## 3.3 Data manipulation: dplyr

The packages *dplyr* and *tidyr* by Hadley Wickham are extremely useful for data wrangling. Here we'll go over a basic intro to *dplyr*. This package helps to manipulate dataframes while reducing repetition.

*dplyr* also uses **pipes** to string together commands. A pipe looks like: %>%

1. *group_by()* groups your data by one or more criteria
   data %>% group_by(Year, Site, Species)

2. *tally()* counts the observations and can often be used in conjunction with *group_by()*
   data %>% group_by(Year, Site, Species) %>%
   tally()

3. *summarize()* can also be used with *group_by()*
   data %>% group_by(Year, Site, Species) %>%
   summarize(mean_height = mean(Height_cm, na.rm = TRUE))

   **Challenge: dplyr 1**
   What happens if we don't include "na.rm = TRUE"?

   **Challenge: dplyr 2**
   Across all years, how many individuals were measured at each site?

4. *select()* keeps only certain variables or columns
   data %>% select(Year, Site, Species)

5. *filter()* keeps only rows that meet certain criteria
   data %>% filter(Year == 1993 | Year == 1996)
   data %>% filter(Year == 1993 & Species == "LATR2")
   data %>% filter(Year == 1993, Species == "LATR2")

   **Note:** you need two equals signs "==" if you want to select a specific numeric value. The | here means "or", the &' means "and' (you can also use a comma,the latter two lines do the same thing). When filtering with factors, the factor levels are character types so need to be in quotes (not going into details now...)

   **Challenge: dplyr 3**
   Subset the data to get rid of the notes column and include only observations from the "rs" site in 1993 without missing data and save it as a new object/variable.

To save a dataset, you can use the *write.table()* function.

**Challenge: writing data**
Save your new dataset using the *write.table()* function. Use help functions if you need!

For more dplyr hints, check out this cheatsheet: `https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf`

## 3.4 Graphics in R

We could dedicate the rest of our lives to this. There are a ton of ways to graphically represent data, here I'll introduce you to graphics in the package {*ggplot2*}. If we need to be able to do certain advanced graphics, we will walk through it. Else, try and use the internet to search for how to make basic graphs!

*ggplot2* using layers that can be added on. Let's walk through a few examples.

The *ggplot()* functionis the base for the *ggplot2* graphics world. We give **global** arguments to this function that apply to all **layers** in the plot. Let's start by telling it what dataset to plot, and what to put on the x and y axes using *aes()* which stands for **aesthetics**.

```
ggplot(data = data, aes(x = Height_cm, y = Big_diam_cm))
```

Nothing happens! We need to tell ggplot how to visually represent the data using a **geom** layer.

```
ggplot(data = data, aes(x = Height_cm, y = Big_diam_cm)) + geom_point()
```

Note: you add layers using the "+" sign. You can continue all code on the same line, or include the "+" on the current line and the next layer on the following line (for ease of reading):

```
ggplot(data = data, aes(x = Height_cm, y = Big_diam_cm)) +
geom_point()
```

Now you should get a nice scatter plot in the "Plots" window. Let's color the points according to sites by adding another aesthetic:

```
ggplot(data = data, aes(x = Height_cm, y = Big_diam_cm, colour = Site)) +
geom_point()
```

Even more exciting, let's add trendlines!

```
ggplot(data = data, aes(x = Height_cm, y = Big_diam_cm, colour = Site)) +
```

```
geom_point() +
stat_smooth(method = lm)
```

Another cool feature is facetting, which makes multi-panel plots based on some factor. Let's see an example:

```
ggplot(data = data, aes(x = Height_cm, y = Big_diam_cm, colour = Site)) +
geom_point() +
stat_smooth(method = lm) +
facet_wrap( Year)
```

Or use a facetted grid to facet by two different factors:

```
ggplot(data = data, aes(x = Height_cm, y = Big_diam_cm, colour = Site)) +
geom_point() +
stat_smooth(method = lm) +
facet_grid(Site Year)
```

**Facet-nating!**

The default is to have a gray background with white gridlines. You can change a lot of things as you see fit using additonal layers, for instance:

```
ggplot(data = data, aes(x = Height_cm, y = Big_diam_cm), colour = Site) +
geom_point() +
stat_smooth(method = lm) +
facet_grid(Site Year)
theme_bw() +
xlab("Plant Height (cm)") +
ylab("Plant Large Diam (cm)")
```

Plots can also be saved as objects, for instance:

```
p1 <- ggplot(data = data, aes(x = Height_cm, y = Big_diam_cm), colour = Site) +
geom_point() +
stat_smooth(method = lm) +
facet_grid(Site Year)
theme_bw() +
xlab("Plant Height (cm)") +
ylab("Plant Large Diam (cm)")
p1
```

You can then save your plots using the *ggsave()* function:

ggsave("plot_name.pdf", p1)

**Challenge: ggplot2**

Make a figure with separate panels for each site with boxplots for plant height throughout the years. Give it an informative title and save it. Hint: you may need to change a datatype!

It really is endless in terms of types of plots (barplots, boxplots, line plots) and aesthetic things, this is just the briefest of introductions. Check out this cheatsheet for more on *ggplot2* `https://www.rstudio.com/wp-content/uploads/2015/03/ggplot2-cheatsheet.pdf` and this for colors in R `https://www.nceas.ucsb.edu/~frazier/RSpatialGuides/colorPaletteCheatsheet.pdf` I personally use *ggplot2* along with the package *cowplot* to produce most of my figures `https://cran.r-project.org/web/packages/cowplot/vignettes/introduction.html`. It's all about personal preferences at some point.

# 4 Project Management

You should save your Rscript for later use with a ".R" extension. RStudio makes this easy, just use File -> Save as. I don't think I've ever saved my workspace image (which RStudio asks you to do when you exit).

Usually you'll have lots of Rscripts, raw and clean data files, tables, figures, etc. for any project you're working on. RStudio has special features great for project management, see more details here:`http://swcarpentry.github.io/r-novice-gapminder/02-project-intro/` since I'm likely already over class time...

# 5 References

Parts of this tutorial were inspired by Dr. Paul Hurtado's tutorial from the 2013 Joint MBI-NIMBioS-CAMBAM Summer Graduate Workshop at the University of Tennessee in Knoxville, TN and various (C) Software Carpentry lessons `https://software-carpentry.org/` licensed under Creative Commons `https://creativecommons.org/licenses/by/4.0/` (changes have been made).


- Hurtado, Paul. 2013. An introduction to programming in R. Joint 2013 MBI-NIMBioS-CAMBAM Summer Graduate Workshop

- R Development Core Team. 2004. R: A language and environment for statistical comput- ing. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org.

- RStudio. 2012. RStudio: Integrated development environment for R (Version 0.96.122) [Computer software]. Boston, MA.